Scalability Problems in Multiprocessors with Private Caches

Michel Dubois, Luiz A. Barroso, Yung-Syau Chen and Koray Oner

CENG Technical Report 92-12

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213)740-4475

Scalability Problems in Multiprocessors with Private Caches

Michel Dubois, Luiz Barroso, Yung-Syau Chen and Koray Öner

Department of Electrical Engineering Systems
University of Southern California
Los Angeles, CA90089-2562

Abstract. This paper addresses the performance problems caused by high memory access latencies and their effect on the scalability of shared-memory multiprocessor systems. We propose to take full advantage of weak ordering through lockup-free caches and delayed consistency protocols and to interconnect processors with point-to-point links. Simulation results show that these approaches are promising.

1 Introduction

One of the most active topics of research in architecture today is the design of scalable shared-memory multiprocessor systems. An architecture scales well if processor efficiency remains good as the number and/or the speed of processors are increased. One major impediment to better scalability is the widening speed gap between processor technology and memory/interconnect technology. While the speed of processors doubles every two to three years, the speed of dynamic RAMs (Random Access Memories) is improving at a much lower pace. A cache is very effective in closing the gap but, as latencies increase, the gap cannot be completely filled because of cache misses.The problem is compounded in multiprocessors for several reasons. First, the latencies are drastically higher in multiprocessors: each processor and its cache must access remote memories; these accesses involve arbitration protocols for shared communication links and transfers on wires. Second, invalidations of blocks or broadcast of writes must be propagated and add to the processor blocking time. Finally, the miss rate is usually higher because of coherence enforcement. Therefore, as more processors are added to a multiprocessor system, the efficiency of each processor goes down; at one point the cost-effectiveness of the system becomes questionable.

The major aspect of shared-memory multiprocessor systems addressed in this paper is the reduction of processor blocking time due to cache misses and coherence activity. We concentrate on write-invalidate cache protocols. We will use the following terminology throughout the paper. *Memory requests* are requests sent by a processor and its cache to the memory system. There are two types of memory requests: *misses* and *invalidations*. A miss occurs when a new block must be loaded into cache (either on a read or on a write access); an invalidation may be sent to other caches having a copy of the block on a write access. The *latency* of a memory request is the time (in processor cycles) taken by the memory request. The *penalty* of a memory request is the time (in processor cycles) that the processor is blocked during the memory request. In most systems penalties are lower than latencies because of access prefetching and buffering in the processor and because loads can be partially overlapped with processor execution of other instructions.

Processors cannot tolerate arbitrary latencies, however large they may be. Nonetheless, we can improve processor efficiency and system scalability by

1. overlapping memory requests with executions of other instructions,
2. overlapping several memory requests together,
3. reducing the number of memory requests, and
4. reducing the latencies of memory requests

in each processor.

The lockup-free cache and delayed consistency protocols of Sections 4 realize goals 1, 2 and 3. To pursue goal 4, we need to design interconnections between processors with short, point-to-point links, very low arbitration

protocol overhead and high bandwidth; the ring architecture, evaluated in Section 5, is an example of such an interconnection. First, we overview memory consistency models and several approaches to latency tolerance in Sections 2 and 3.

2 Memory Consistency Models

The *memory consistency model* refers to the logical model offered by the memory system to the programmer or to the compiler. This model in turns constrains the possible ordering and interleaving of memory accesses in the multiprocessor. *Sequential consistency* is logically the strongest model and the most restrictive as far as the ordering of memory accesses is concerned; it requires that memory accesses be *strongly ordered*. A sufficient condition for strong ordering is that all memory accesses from processors are *globally performed* in the program order of each processor [1, 2, 3]. Broadly speaking, a store is globally performed when all processors can observe the new value, and a load is globally performed as soon as the store creating the value read has been globally performed.

The major drawback of strongly ordered memories is that they prevent the effective use of store buffers. In Fig.1, a processor is displayed with its store buffer. (The processor may also have a first level write-through cache.) In a strongly ordered memory system, a load hitting in the cache cannot be globally performed in the cache before all previous stores in the buffer have been globally performed. This restriction makes the store buffer all but ineffective.

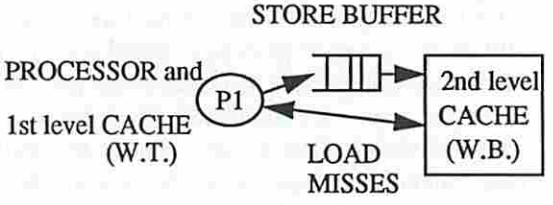

Fig. 1. Processor with caches and store buffer (W.T.: Write-Through; W.B.: Write-Back)

To remedy this problem it has been proposed to relax the constraints on the ordering of shared memory accesses [1] by assuming that all synchronizations among parallel threads must be done through explicit, hardware-recognizable synchronization operations (i.e., these operations must be distinguishable from regular load/store instructions.) Strong ordering is only enforced on these synchronization operations rather than on all memory accesses. The order of other memory accesses must be such that

- they respect intra-processor dependencies, and
- they respect the partial order imposed by explicit synchronization operations. A processor must perform all its preceding loads and stores globally before it can issue a synchronization operation; moreover, a processor may issue no memory load or store following a synchronization point until the synchronization operation is successfully completed.

In systems where these two conditions apply we say that loads and stores are *weakly ordered* and that the memory system is weakly ordered.

Special types of synchronization operations may allow additional relaxation of the above conditions. For example, for synchronization based on critical sections, a refinement called *release consistency* [4] distinguishes between locking (acquiring) a lock and unlocking (releasing) a lock. In this case the second condition above becomes:

- all stores preceding a lock_release must be globally performed before the release, and no load following a lock_acquire in the code may be issued before acquiring the lock.

In weakly ordered systems the processor is not blocked when a store misses in the cache or when it requires an invalidation: stores are buffered and load hits can by-pass the stores in the buffer; the only requirement is that the store buffer must be emptied before any synchronization operations can be executed. The savings can be significant: if the fraction of stores in a program is 10% and if the memory request rate (misses and invalidations) for such stores is 5% the number of cycles saved per instruction can be up to 0.5% of the memory request latency. For latencies

between 20 and 100 machine cycles the savings in execution times can be as high as 10 to 50%. To reap these performance gains, the processor must have a first level cache to satisfy loads. This first level cache is usually present in current microprocessor chips in the form of a 4 to 16 kbytes write-through data cache.

When synchronizations are done through critical sections, a slight additional performance gain is possible for the system of Fig. 1 by adopting release consistency. Namely, a lock_release can be considered as a store and inserted in the store buffer. Moreover, the store buffer does not have to be emptied on a lock_acquire. These possible performance gains depend on the size of the store buffer and on the frequency of lock_acquire/lock_release operations and are not as large as the possible gains between strong and weak orderings [5].

3 Overlap Between Memory Requests and Processor Execution

Penalties can be reduced by overlapping one memory request with processor execution and/or by overlapping several memory requests together.

3.1 Overlapping one Memory Request with Processor Execution

In this case, only one memory request can be in progress at any time. While the memory request is serviced the processor continues execution. The processor is blocked when it needs to send a second memory request while one is in progress. Fig. 2 illustrates the possible gain.

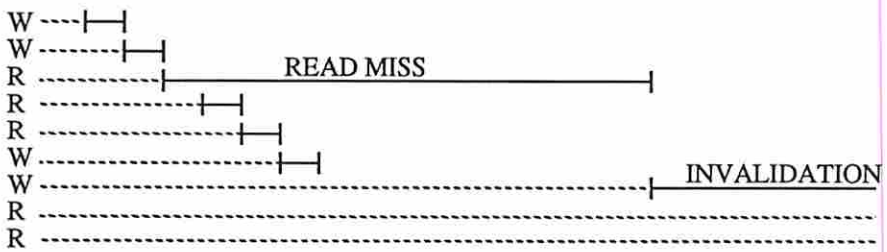

Fig. 2. Overlap of one miss with processor execution

Key to the performance of such systems is the ability to execute the maximum number of instructions while a memory request is pending.

3.2 Overlapping Several Memory Requests Together

Larger performance gains are possible if several memory requests can be overlapped with execution. The modified timing diagram for the sequence of accesses used in Fig. 2 is shown in Fig. 3. For memory systems with very large

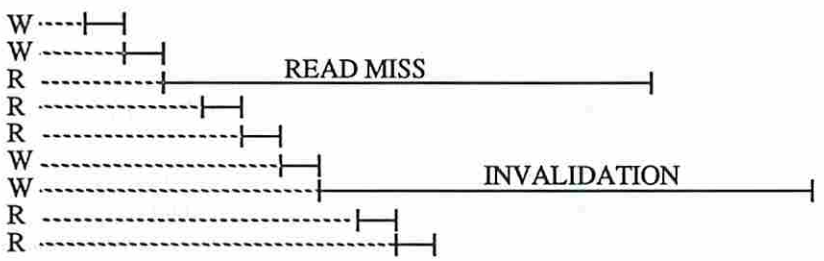

Fig. 3. Overlap of two memory requests

latencies, memory request penalties (rather than instruction execution) dominate the execution time and thus the issuance of memory requests in parallel is the only effective way to hide the latencies.

3.3 Memory Request/Processor Execution Overlap in The DASH Protocol

The DASH prototype built at Stanford has limited capability to overlap memory requests with processor execution.

Each processor in the DASH has the architecture shown in Fig. 1. Published information about the DASH protocol [4] indicates the following:

- When a load or store miss occurs in the first and second level caches the processor is blocked.
- When an invalidation is sent by the second level cache (without a miss), the write buffer is blocked but the processor is not.
- When the store buffer is full the processor blocks on a store instruction.

From these constraints it appears that the DASH protocol can only overlap invalidations (without miss) with processor execution. It cannot overlap misses in the second level cache with processor execution nor overlap several misses and invalidations. This is probably why the simulations of the DASH machine show little difference in performance between strong and weak ordering.

3.4 Multithreaded Architectures

A simple idea derived from the HEP architecture [6] consists of running different threads in multiple contexts resident in each processor and of switching from one thread to the next at the occurrence of a memory request [7, 8]. The advantages of this solution are:

- Simpler compilers: there is no need for compilers to move the code to maximize overlap.
- Strong ordering (and therefore sequential consistency) can be efficiently enforced in each thread with no need to weaken the consistency model. (Sequential consistency is really desirable, from the programmer point of view.)
- Larger and larger latencies can be tolerated by simply increasing the number of contexts.

However, there are several problems with this approach.

- Special processors must be designed for multiprocessors, with several processor-resident register sets.
- Precious real estate on the processor chip is used for a large number of registers. At any time only one fraction of these registers (one context) is active.
- Switching from one thread to the next is complex for high-performance uniprocessors such as superpipelined and superscalar machines [9]. Many cycles are needed for thread switching. (This is a fixed and hard-to-reduce penalty.)
- The hit rate in the cache is reduced unless the threads running concurrently on a processor share a lot of data.
- Programs with limited parallelism cannot really take advantage of the mechanism.
- The mechanism does not speed up each individual thread (actually it tends to slow them down) and therefore it is not good for single thread programs or programs with serial bottlenecks [10].

While multithreaded architectures are an interesting approach to latency tolerance in large-scale systems their promise is still to be realized. In the following we explore further the possibilities offered by weak ordering in systems running one thread per processor.

4 Taking Full Advantage of Weak Ordering: Lockup-free Caches and Delayed Consistency.

To take full advantage of weak ordering the cache must be non-blocking for load and store misses and for invalidations. There must be enough hardware in the processor node to support multiple pending memory requests. Since [11] it is known that caches which do not lock out the processor on store or load misses can be designed for weakly ordered systems. (Indeed, unblocking the cache on misses is equivalent to performing the corresponding loads and stores out of order.) Moreover, in order to support multiple pending invalidations, modified blocks for which ownership has not been acquired must be buffered, leading to *delayed consistency* protocols. To simplify the discussion we describe and evaluate non-blocking caches and delayed consistency separately.

4.1 Lockup-free Caches

Kroft described a uniprocessor lockup-free (or non-blocking) cache in [12]. In his design, multiple misses are resolved

concurrently by storing information about each miss in MSHRs (Miss Status Holding Registers) and then forwarding the miss request, packaged along with vital return information, to the main memory. There can be *primary* or *secondary* misses: the first pending miss on a block is called the primary miss. A secondary miss results from a load or store to a block for which there is already a pending primary miss; a secondary miss is most likely to be a hit in the case of a blocking cache and should not trigger a memory request in the case of a lockup-free cache.

In-cache Versus In-register Loads

A processor can utilize a lockup-free cache if a read or write miss does not cause the entire processor to block. RISC architectures with non-blocking loads work well with lockup-free caches. We distinguish between two types of non-blocking loads: *in-cache* loads and *in-register* loads. In-cache loads do not specify a target register and simply load the data in cache; they are use to prefetch data blocks in the cache. In-cache prefetching requires that the addresses of all prefetched data be computed twice (once for the prefetch and once for the load in register), consuming processor cycles and registers; accesses that miss must be predicted, otherwise a large number of the prefetches are useless and degrade performance. Moreover, prefetched blocks may be replaced before being used. In-register loads load the data in a register and therefore do not have these drawbacks. However, the number of pending in-register loads is limited by the number of available registers. Moreover, hardware complexity goes beyond simply adding a non-blocking load instruction: when an in-register load misses in the cache, the processor executes subsequent instructions in parallel with the load miss; later, when the load completes, a low-level trap must interrupt the processor and directs it to load the value in a specific register.

Combined Non-blocking Processor/Cache Architecture

One problem with lockup-free caches is their complexity. This complexity can be totally eliminated by combining the non-blocking mechanism in the processor and in the cache. The following is a combined processor and cache architecture with a simple and efficient mechanism to support both in-cache and in-register non-blocking loads. An interesting feature of the architecture is the way in-register load misses are dealt with. When an in-register load misses in the cache, the address of the word is loaded in the register and the register is locked (using a one-bit tag such as the tag used to detect hazards on registers [13]). The subsequent instruction reading the data blocks the processor and the address in the register is then used to access the cache. The cache algorithm works as follows:

- Primary miss: a cache block frame is allocated (implying a possible replacement) and the state of the block is set to pending. A pending block cannot be replaced. If the access is an in-register load, the address is returned to the processor and stored in the register; the register is tagged as busy. In all cases, the block is loaded in cache from memory and, while the miss is pending, the processor and the cache are not blocked.
- When an instruction reads a busy register an implicit load is triggered using the address in the register. An implicit load always blocks the processor, the point being that the data is needed to complete the instruction execution.
- When the implicit load completes (whether it missed or not) the register is loaded with the data value and the register is unlocked.
- Secondary miss: a secondary miss is detected when the accessed block is pending in the cache. In the case of an in-register load the address is stored in the register and the register is tagged busy. No request is sent to memory. The cache and the processor remain unblocked.

In this design, we do not need to keep track of pending loads and their target registers in MSHRs, which limit the number of pending misses in Kroft's design. Secondary store and load misses are handled automatically. Load completion traps are eliminated.

Effect of Dependencies

Control and data dependencies limit the amount of overlap. A control dependency occurs when a conditional branch instruction tests the result of a previous instruction. Conditional branch instructions and their target instruction partition the program into *basic blocks*. It is usually very difficult to move a load outside of a basic block. Data dependencies affecting the possible overlap between loads and execution of other instructions are true, read-after-write

dependencies on the registers which are targets of loads. A true dependency exists between a load instruction and the set of instructions which subsequently read the register. We define the *dependency distance* of a load as the minimum number of processor cycles between the load and the first following instruction which reads the target register (either another load or a register-to-register instruction). Compilers must modify the code to increase the size of basic blocks and then hoist loads as far up as possible. These two phases can be implemented most easily in scientific code. In the case of Fortran DO-loops, hoisting loads beyond basic blocks is equivalent to pipelining iterations of the loops.

Lawrence Livermore Loop Simulations

We have run simulations of the combined cache/processor architecture for the Lawrence Livermore Loops [14]. Some results are displayed in Fig. 4 for Loop 1 and in Fig. 5 for Loop 9. To derive these curves, we first compiled the program into SPARC assembly code, then hoisted the loads manually across iterations of the loop. This approach is equivalent to pipelining at the assembly code level: the operands needed in iteration i are fetched in iteration i-k where k is the number of pipeline stages. The execution of the code was simulated on an instruction level simulator of a SPARC engine with 32 registers, an 8kbyte direct-mapped cache with 32 byte blocks and an interleaved memory with infinite number of memory banks (zero conflict). Stores were buffered in an infinite size store buffer so that their penalty is negligeable, leaving only load miss penalties.

```
      subroutine Loop1(990,Q,R,T,X,Y,Z)
      integer n,k
      double precision Q,R,T,X(1001),Y(1001),Z(1001)
      do 1 k = 1,990
1           X(k) = Q + (Y(k) * ((R * Z(k+10)) + (T * Z(k+11))))
      return
      end


      subroutine Loop9(101,CO,DM22,DM23,DM24,DM25,DM26,DM27,DM28,PX)
      integer i,n
      double precision CO,DM22,DM23,DM24,DM25,DM26,DM27,DM28,PX(25,101)
      do 9 i = 1,101
            PX(1,i) = PX(3,i) + (CO * (PX(5,i) + PX(6,i))) +
     .          (DM28 * PX(13,i)) + (DM27 * PX(12,i)) +
     .          (DM26 * PX(11,i)) + (DM25 * PX(10,i)) +
     .          (DM24 * PX(9,i)) + (DM23 * PX(8,i)) +
     .          (DM22 * PX(7,i))
9     continue
      return
      end
```

Fig. 4. Fortran programs for Loops 1 and 9

In Loop 1, Y(k), Z(k+10) and Z(k+11) are fetched in each iteration. The maximum number of pending memory accesses is 3 if they all map to different cache blocks and if all accesses miss. As we use single precision floating point numbers and as the block size is 32 bytes, one block can hold up to 8 words of one array. Most of the time accesses to Z(k+10) and Z(k+11) map to the same cache block and only one memory request is needed for both of them. Therefore on the average there will be 2 pending memory accesses at any time.

In an iteration of Loop 9 there are 10 reads and 1 write. In one iteration different elements in one column of 2 dimensional matrix PX are read, PX(1,i) first and PX(13,i) last. If PX(1,i) maps to one of the last words in one cache block, successive cache blocks must be accessed during one iteration. Thus there may be 3 pending memory accesses at one time. When a 1 stage pipeline is applied to the code this number will be doubled as 2 iterations of the loop are in progress at any time. Each additional stage of pipeline adds 3 accesses.

Without code hoisting, the lockup free cache is totally ineffective. When the code is hoisted and more pipeline stages are applied to the loop, the execution time is reduced and the speedup (defined as the execution time on a system with blocking loads divided by the execution time on a system with non-blocking loads) increases as shown in Fig. 4.

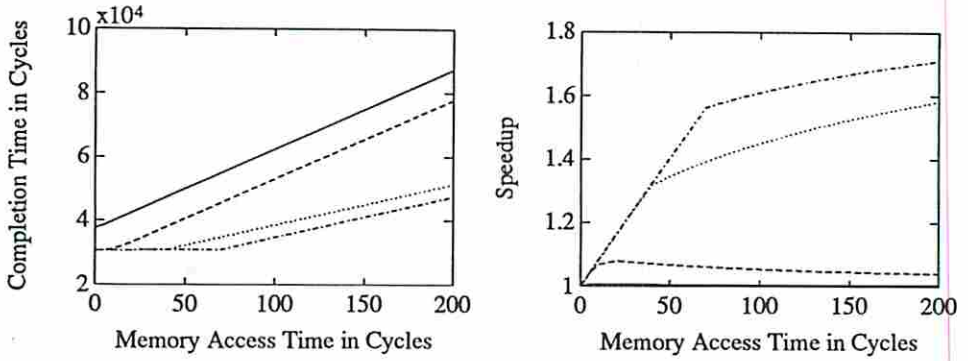

Fig. 4. Effect of load hoisting and pipelining on the latency tolerance of Loop 1
Solid line: No load hoisting. Dashed line: Load hoisting, 0 pipeline stage.
Dotted line: 1 pipeline stage. Dashdot line: 2 pipeline stages.

For low memory latencies, the execution time curve is almost flat and the speedup increases roughly linearly. The reason is that the memory latency is less than the dependency distance of most loads. For these latencies we say that the program is *latency tolerant*. Note that the program still benefits from the overlap even if the memory latency is larger than the maximum dependency distance of a program.

The above example would seem to indicate that any amount of memory latency can be tolerated through pipelining provided the loop has enough iterations. Unfortunately, a major limitation is the number of registers. The deeper the software pipeline, the more registers we need as we allocate one register for each pending load. For example, in the case of Loop 9, a one stage pipeline consumes all 32 registers.

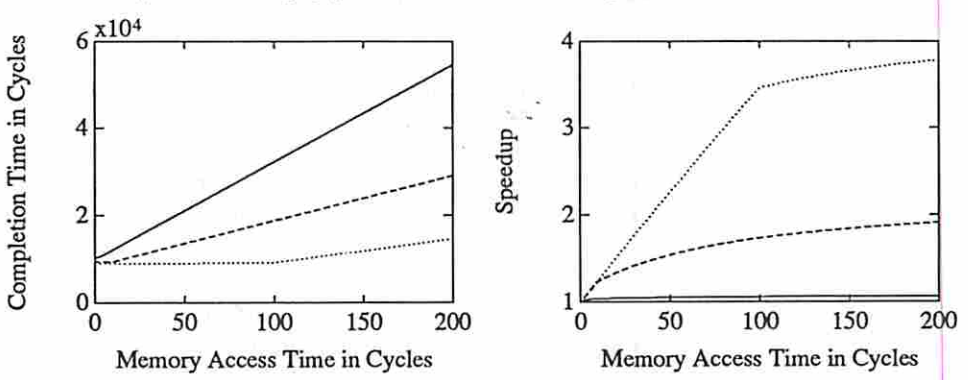

Fig. 5. Effect of load hoisting and pipelining on the latency tolerance of Loop 9
Solid line: No load hoisting. Dashed line: Load hoisting, 0 pipeline stage.
Dotted line: 1 pipeline stage.

Discussion

The speedup curves of Fig. 4 and 5 exhibit different behaviors for low and high memory latencies.These behaviors can be explained by the following model.

Let T be the miss latency and T_d be the average dependency distance. We distinguish between $T \ll T_d$ and $T \gg T_d$.

Case 1: $T \ll T_d$

In this case, the dependency distances are long enough that all load misses are overlapped with execution. Therefore,

if T_{ex} is the execution time of the program when all loads hit in the cache and if M is the total number of load misses, the execution times are $T_{ex}+M\times T$ and T_{ex}, for the blocking and the non-blocking cache respectively. Therefore the speedup is:

$$Sp = \frac{T_{ex}+M\times T}{T_{ex}} = 1+M\times T^0 \qquad with\ T^0 = T/T_{ex}$$

Case 2: $T \gg T_d$

In this case, most dependency distances are too short to cover the miss latency and the penalty of the non-blocking cache is $T-T_d$ if we assume that one single load miss can be pending at any one time. However, this penalty is further reduced because of the overlap among multiple load misses. We define f, the memory access overlap factor, as the average number of primary load misses pending whenever the processor is blocked. The average penalty of a load miss in the lockup-free cache is $(T-T_d)/f$ and the execution time is $T_{ex}+M\times(T-T_d)/f$; therefore, the speedup is

$$Sp = \frac{T_{ex}+M\times T}{T_{ex}+M\times(T-T_d)/f} = \frac{1+M\times T^0}{1+M\times(T^0-T^0{}_d)/f}$$

As T becomes very large (with respect to T_d) the speedup tends to f, the memory access overlap factor. Fig. 6 shows

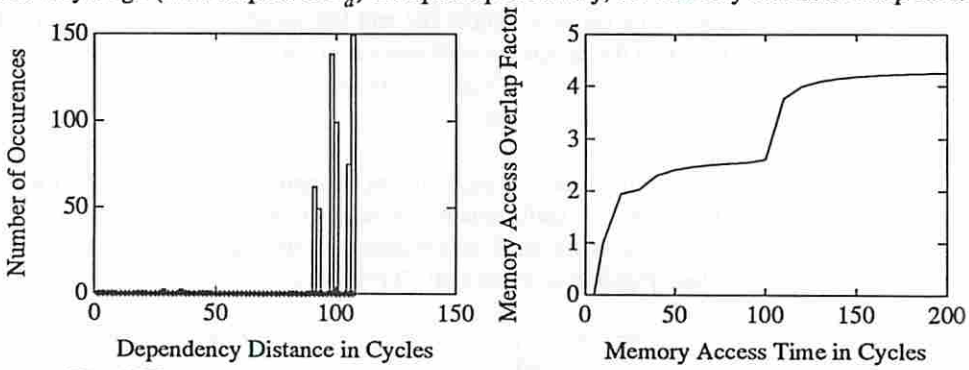

Fig. 6. Histogram of dependency distances and memory access overlap factor for Loop 9 (1 pipeline stage)

the histogram of dependency distances and the memory access overlap factor for Loop 9 with 1 pipeline stage: both the speedup and the memory access overlap factor tend to 4 for high latency. The maximum dependency distance is around 100 which is also the latency for which the speedup curve behavior changes. The same observation would apply to all cases.

Therefore, we can say that increasing the dependency distance is effective for hiding moderate memory latencies. However, for very large latencies, overlapping multiple memory loads is even more effective. In addition to increasing dependency distances, a good compiler should also group loads together so that a dependency does not block the processor and prevents it from issuing more loads.

In parallel code, hoisting loads is more difficult than in serial code, especially for DOACROSS loops [15]. It is unsafe to move an in-register load up before a preceding lock operation in the code because the registers are not subject to the cache coherence protocol. An interesting approach is to move in-register loads as high in the code as possible but no higher than a preceding lock operation and to prefetch in the cache for the loads that are most likely to miss (miss prediction can be done using Porterfield's overflow iteration idea in [16].) The prefetch in cache can be implemented with an in-cache load which is safe since the prefetched value is subject to the cache coherence protocol.

4.2 Delayed Consistency

In all existing multiprocessors coherence is enforced as soon as possible (on the fly). There are two drawbacks to On-The-Fly (OTF) write invalidate protocols: high invalidation penalties and high false sharing miss rate. (False sharing is the sharing of blocks without actual sharing of data [17].) On the other hand, in delayed protocols [18], some coherence actions such as the propagation of invalidations are deliberately delayed and multiple, inconsistent copies

of the same block can exist at any time. Delaying invalidations increases the concurrency of accesses to blocks and permits more flexible overlap between the execution of a processor and the propagation of several of its invalidations. Processor efficiency is improved at the cost of slightly more complex hardware.

In an OTF protocol, stores into non-owned blocks require the sending of invalidations; later these invalidations are received by other cache controllers, which invalidate their copy. Each of these two phases (the sending of invalidations and the invalidation of copies) can be delayed. A protocol which delays the sending of invalidation is called a Send Delayed (SD) protocol; a protocol which delays an incoming invalidation is called a Receive Delayed (RD) protocol. Finally, a protocol which delays invalidations at both ends is called a Send-and-Receive Delayed (SRD) protocol.

When a cache controller has to send an invalidation to other caches following a local store, the invalidation can be buffered in a local buffer called the Invalidation Send Buffer (ISB) until the next lock_release in the processor. Similarly an incoming invalidation can be buffered in the Invalidation Receive Buffer (IRB) until the next lock_acquire in the processor. Fig. 7 shows an architecture with store buffers and invalidation buffers.

Receive Delayed (RD) Protocol

The function of the IRB is to let the processor access the block even if an invalidation has been received. This is safe because the programmer of a weakly-ordered system can make no assumption about the relative order of shared data accesses, except at synchronization points. If an invalidation is received and buffered and if a load is performed on the same block in the local cache, we can see this occurrence as a form of prefetching (the load returns the value defined before the invalidation.) Therefore, a received invalidation can be delayed, but only until the next lock_acquire (because no value fetched after the acquire in the code can be prefetched before the acquire is successfully completed.) In practice, the IRB should not be a real hardware buffer because, if it were, the lock_acquire would be very slow; moreover, the number of pending invalidations would be limited by the size of the buffer. Rather, we have proposed [18] to implement the IRB with a additional bit in the cache directory, the Stale bit. When an invalidation reaches the

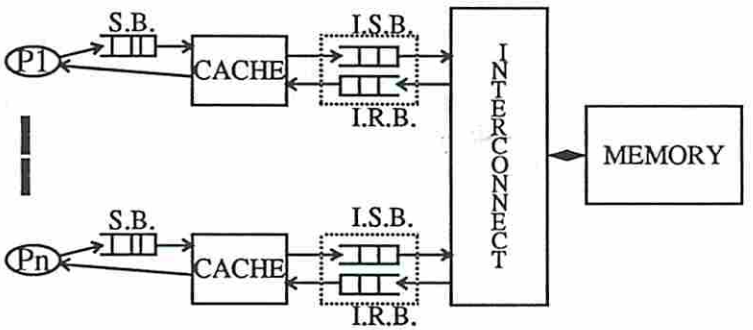

Fig. 7. Multiprocessor with Store Buffers (SB) and Invalidation Buffers (IRB and ISB)

cache it sets the Stale bit of the block. A stale block can be read but cannot be written into before ownership has been acquired. All stale blocks become invalid in the cache right after a lock_acquire. An RD protocol does not increase the overlap between memory requests and processor execution but it reduces the false sharing miss rate.

Send-and-Receive Delayed (SRD) Protocols

A Receive Delayed protocol can also be Send Delayed by buffering outgoing invalidations in an Invalidation Send Buffer (ISB). The ISB is a small, fully associative hardware buffer capable of holding a few cache blocks. When a store occurs on a non-owned copy a blockframe is allocated in the local ISB to store the value. Consecutive stores to a block by the same processor are buffered in the ISB. According to the release consistency model, all entries in the ISB must be removed at the next lock_release[1]. An entry must also be removed from the ISB every time a new ISB entry must be allocated. When an entry is removed from the ISB an invalidation is sent and memory is partially updated (only the modified items in the block are sent to memory); if the block was stale then ownership is not acquired and

1. Additionally, the lock_release could also be buffered in the ISB. We have not tried this.

the block remains stale in the cache; otherwise ownership is acquired for the block. During these operations neither the cache nor the processor are blocked. The details of this protocol are given in [18].

The SRD protocol further reduces coherence activity due to false sharing and also permits overlap between processor execution and propagation of invalidations. In our simulations we block the cache on a miss. However a straightforward extension of the protocol would avoid blocking time on store misses: a store miss could simply fill a word in an allocated block of the ISB.

Simulation Methodology

We have done extensive simulations of the three protocols (OTF, RD, and SRD). The OTF protocol is Censier and Feautrier's directory-based protocol [19]; and the RD and SRD protocols are derived from it. Details of the protocols can be found in [18].

In order to evaluate the performance of the three protocols, we have built a set of trace-driven simulation models of the three cache coherence protocols. In the simulator trace records are taken one by one in the order that they appear in the trace file. At each reference, the cache directories are modified according to the protocol specifications and event counts are updated. All simulations assume 128 Kbyte direct-mapped data caches with 16 to 128 byte blocks and separate instruction caches. Since the hit rate for instructions was observed to be extremely high, we made the simplifying assumption that instructions have a hit rate of 100%. This assumption introduced negligible effects in all output parameters but greatly reduced the elapsed time of each run of the simulation. In the SRD protocol, an ISB of size two blocks was adopted, since it is sufficient to remove all false sharing misses in the three traces we ran.

The three benchmarks used in our evaluation (FFT, SIMPLE and WEATHER) were obtained from Anant Agrawal's group at MIT and are available to any one wanting to verify the results of this paper. They are 64-processor parallel FORTRAN programs traced using a postmortem scheduler (which is a method that derives parallel traces from a uniprocessor execution with embedded synchronization information). The FFT is a radix-2 Fast Fourier Transform program. SIMPLE solves equations for hydrodynamics behavior using finite difference methods. WEATHER also uses finite difference methods to model the atmosphere around the globe. There are four types of accesses in the traces: accesses to instructions, accesses to private data, accesses to shared data and accesses to synchronization variables. The synchronization accesses are Fetch-and-Adds. In the simulation, synchronization variables are not loaded in cache. Since we do not know the nature of the synchronization, we consider all Fetch-and-Adds as lock_acquire directly followed by lock_release. The main characteristics of the non-synchronization accesses in the traces are shown in Table 1.

Table 1: Trace Characteristics (ref. in 1000's)

Trace	Data references	Instruction fetches	Local data references	Shared data references
FFT	4,313	3,124	3,279 (26.6% write)	1,034 (49.9% write)
SIMPLE	14,018	11,594	9,936 (34.7% write)	4,074 (10.9% write)
WEATHER	15,628	13,638	13,110(15.7% write)	2,518(19.2% write)

Discussion

Fig. 8 and 9 show the protocol effects on the miss rates for FFT and WEATHER and for block sizes between 16 and 128 bytes. The total miss rate on data blocks has three components: the false sharing miss rate, the rate of other misses to shared blocks (i.e., cold misses, replacement misses and true sharing misses), and the miss rate to private data. Miss rates are computed as the ratio of the number of misses and the total number of data references (private and shared). The false sharing misses were detected by a technique introduced in [17]. The FFT trace has a large amount of write sharing and all the coherence activity is due to false sharing. The SRD protocol is extremely effective at reducing the

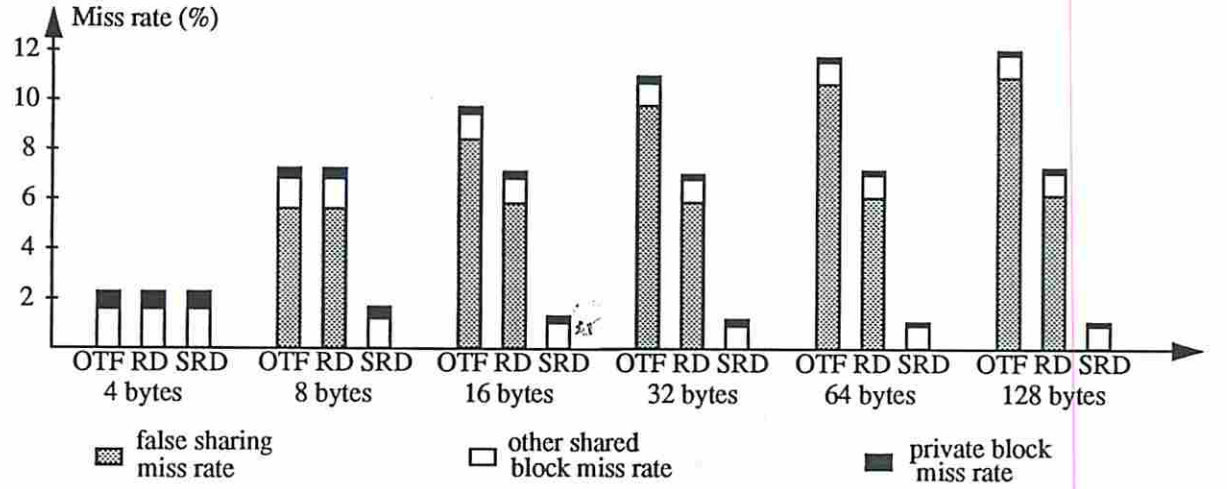

Fig. 8. Miss rates for the FFT trace

false sharing miss rate to zero. The number of other misses decreases slightly with the block size, but this effect is dwarfed by the effect of false sharing misses. The WEATHER trace exhibits some false sharing albeit not as much as the FFT trace. The delayed protocol had very little effect on the miss rate of SIMPLE because the miss rate of SIMPLE is due exclusively to shared data misses (about 16%) but the false sharing miss rate is very small, between 0.5 and 1%.

In all cases the SRD protocol eliminated virtually all the false sharing misses without increasing the number of other misses or of other memory transactions. In the FFT traces the rate of invalidations sent by each processor is

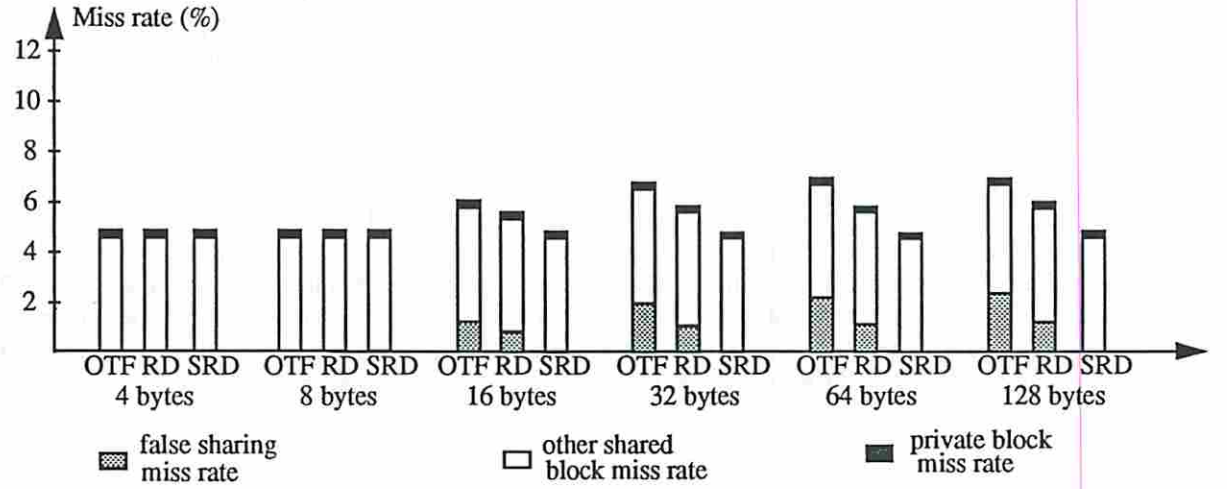

Fig. 9. Miss rates for the WEATHER trace

around 6% (per data reference), for all block sizes and for all protocols. However, in the case of the OTF and RD protocols, all invalidations are issued to request ownership, while in the case of the SRD protocol only 4-10% of the invalidations are issued to request ownership. The situation is similar in the WEATHER trace and in the SIMPLE trace: same number of invalidations for all protocols but reduced number of ownership requests for SRD. Note that

SRD protocols have the added advantage that multiple invalidations can be overlapped in each processor.

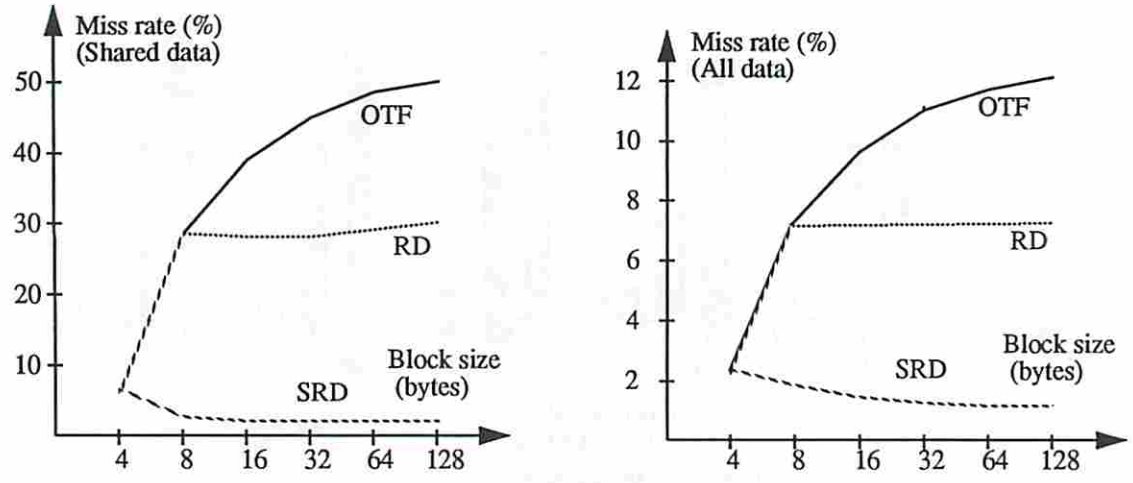

Fig. 10. Miss rates on Shared data and on all data for the FFT trace

5 The Slotted Ring: A High-Bandwidth Cache-Coherent Architecture

After having considered techniques to tolerate high memory request latency we now address the problem of keeping latencies low as the computational power (number of processor and single processor MIPS rate) increases. Systems based on shared buses, which are dominant in today's multiprocessor market, are not likely to show much improvement in terms of communication bandwidth. The clock speed of a bus is limited by propagation delays and signal reflections. On the other hand, point-to-point interconnections lack most of the inherent problems observed in buses, since signal reflections can be eliminated. The SCI Standard proposal [20] is the most notable example of the interest for point-to-point connections in shared-memory multiprocessor systems. It is based on 500 Mbps 16-bit wide unidirectional point-to-point connections.

The Slotted Ring multiprocessor and cache coherence protocol [21] were proposed as an efficient way to make use of the bandwidth delivered by point-to-point interconnections. In the Slotted Ring, point-to-point connections are used to build a very fast pipelined ring interconnection network in which messages are inserted in pipeline slots and circulate through the ring without being removed by any node other than the destination. Also in [21] we demonstrated how a snooping cache coherence protocol, typically used in bus based systems, can be implemented efficiently in the Slotted Ring architecture. Even though rings are usually perceived as high latency topologies, we show in this Section how the Slotted Ring can outperform very fast buses for various system sizes.

5.1 The Slotted Ring Architecture

The Slotted Ring architecture is schematically shown in Fig. 10. Each processor node consists of one RISC processor,

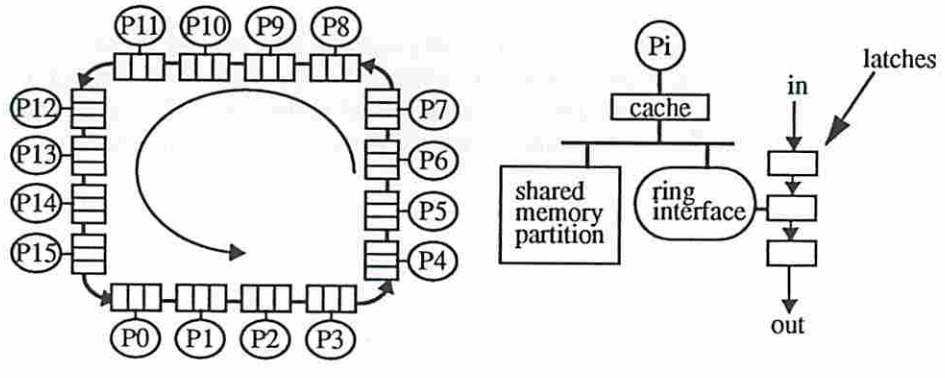

Fig. 10. The Slotted Ring Architecture and processor node

a large private cache, a fraction of the system's physical memory space and the ring interface. The Slotted Ring interconnection network can be viewed as a circular pipeline, with 3*P stages, where P is the number of processor nodes in the system, i.e., each processor node is connected to three 64-bit wide latches in the communication path. At each ring clock cycle, the contents of each latch is copied into the next one. The circular pipeline is statically subdivided in message *slots*, which are groups of one or more pipeline stages, depending on the type of message slot. Messages are inserted in the ring by loading a slot with useful information and removed by marking a slot as empty. A processor node may use the first empty slot that passes through it to send a message, provided the slot type matches the message type. In the Slotted Ring several messages can be transmitted at the same time, which increases the communication bandwidth.

Routing in this architecture is very straightforward. An incoming message is either consumed by a node or forwarded to the next node in the ring order. The ring interface has a very fast logic that scans the first bits of each message and determines whether it has to be forwarded or removed from the ring. A 2-way interleaved dual-directory serves memory requests coming from the ring; it contains a copy of the local cache directory (tags + state information). Interleaving is necessary to allow the ring interface to keep up with the maximum rate in which requests arrive from the ring.

5.2 Snooping on the Slotted Ring

The snooping cache coherence protocol for the Slotted Ring, presented in [21], is a write-invalidate, write-back protocol logically similar to an ownership-based snooping protocol for a split transaction bus. Three cache states, *Invalid* (INV), *Read-Shared* (RS), and *Write-Exclusive* (WE), indicate whether the block is not present in the cache, present in read-only mode, or present in read-write mode respectively. A *dirty bit* is stored with the block frame in memory (i.e. with the home node) to indicate when a block is cached in WE state in some node. The dirty bit is very important to the snooping implementation on the Slotted Ring since it allows the home node to determine if it has the most recent (or valid) copy of a block, and therefore should respond to misses for that block. When the dirty bit is set, the node that has the WE copy of the block (the dirty node) is responsible for responding to memory requests, instead of the home node.

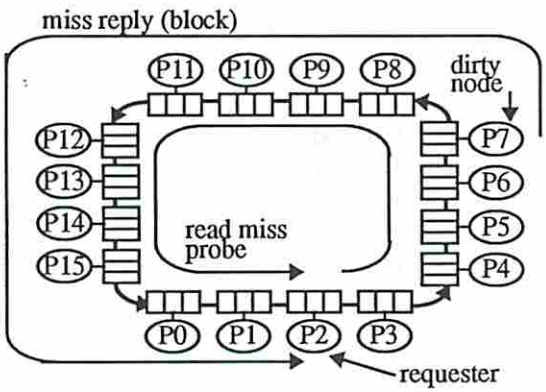

Fig. 11. A read miss transaction in the Slotted Ring under snooping

Read and write miss requests, as well as invalidation requests are broadcasted through the ring, so that all the nodes in the system can snoop on them. We call these memory requests *probes*. Probes are inserted and removed by the issuing node. The node that has the valid copy of a given block (the valid node) is the one that acknowledges probe messages either by issuing a piggyback signal on a later message or by providing a copy of the block to the requesting node. An example of a read miss transaction is shown in Fig. 11. The protocol is a write-invalidate protocol and is

briefly described on Table 2. Further details of the snooping protocol implementation can be found in [21].

Table 2: Snooping Protocol for the Slotted Ring

Transaction	Protocol Behavior

Read Miss	if <block is dirty>
	  dirty node sends copy to requester & changes to RS
	  requester receives block & forwards copy to home node
	  home node resets dirty bit
	else
	  home node sends copy to requester
	requester final state is RS

Write Miss	if <block is dirty>
	  dirty node sends copy to requester & changes to INV
	else
	  all caches with RS copy change to INV
	  home node sends copy to requester & sets dirty bit
	requester final state is WE

Invalidation	/* block is already cached in RS state at requester */ all nodes with RS copies change to INV home node sets dirty bit

The most important feature of this snooping protocol is that all coherence transactions are completed without requiring more than one "trip" around the pipelined ring. In this sense, the latency of coherence transactions is minimum on the snooping protocol. Another interesting feature of this protocol is that the latency involved in satisfying misses is independent of the relative positions of the requesting node and the node with the valid copy of the block.

5.3 Simulation Methodology

In order to evaluate the performance of the Slotted Ring architecture with respect to state-of-the-art bus technology, we have built a set of trace-driven simulation models of the two architectures. Our simulation programs were implemented using a library of process-oriented simulation routines called CSIM [22]. The versatility of CSIM allowed us to construct very detailed cycle-by-cycle simulations of the various systems, accounting for the effects of contention for communication resources and synchronization. The main output parameters obtained are:

- normalized program execution time[2]
- average processor utilization
- average interconnection (ring/bus) utilization
- average remote miss delay (delay of misses that require ring/bus transaction)

The traces used in this analysis are the same ones described in Section 4. We assume that each node has a 128 Kbyte direct-mapped data cache with block size of 16 bytes. As in Section 4, we also assume that instruction fetches always hit in a separate instruction cache.

5.4 Slotted Ring vs. Shared Bus

Now we present some simulation results that compare the performance of the Slotted Ring and a Shared Bus system, both using a snooping protocol. The FFT was the only benchmark that could be used to simulate systems with less than 64 nodes, since it exhibits a very clear barrier synchronization pattern that allowed us to schedule multiple threads per

2. Actual execution time divided by the execution time assuming a 100% hit rate.

node. The processors speed was set to 20 MIPS, which is a somewhat conservative value. In Table 3 below are the basic system parameters assumed in our comparative analysis. The bus parameters are compatible with a split transaction version of the FutureBus+ standard.

Table 3: Bus and Ring Parameters

Bus	Ring
Bus clock: 25 MHz (40ns)	Ring clock: 200 MHz (5ns)
Bus width: 64-bit data paths	Ring links: 64-bit unidirectional
Packed switched bus: snooping protocol	Snooping protocol
Arbitration overlapped with transfer	Number of latches per node: 3

We also assume that the private caches can respond to a hit in one processor cycle, wether it is a write or a read. The delay to fetch a block from a local memory bank was set to 140ns. It should also be taken into account the fact that, in the FFT trace, the miss rate for shared data decreases drastically as the number of processors decreases - from 27% (64 processors) to 2.7% (8 processors). That happens because, in this algorithm, each thread has a higher degree of sharing with its immediate neighbors than with other threads. We exploited this characteristic by statically scheduling neighboring threads of the original trace to the same node, when the number of nodes was less than 64. This allocation resulted in less false sharing misses for the configurations with less than 64 nodes.

Tables 4 through 7 show the simulation results using the FFT trace.

Table 4: Normalized Execution Time - FFT

# processors	8	16	32	64
Ring	1.03	1.06	1.69	3.25
Bus	1.18	1.78	18.43	53.0

Table 5: Average Processor Utilization (%) - FFT

# processors	8	16	32	64
Ring	95.8	94.3	59.7	30.9
Bus	84.2	56.6	6.91	1.91

Table 6: Average Bus/Ring Utilization (%) - FFT

# processors	8	16	32	64
Ring	1.5	3.7	30.1	41.5
Bus	43.3	77.0	98.5	99.7

Table 7: Average Remote Miss Delay (in ns) - FFT

# processors	8	16	32	64
Ring	275	399	669	1,315
Bus	1,773	5,703	21,778	52,144

Tables 8 and 9 show the simulation results for SIMPLE and WEATHER, using a 64 node system. The total

Table 8: SIMPLE - 64 nodes

parameter	Normalized Exec. Time	Processor Utilization	Bus/Ring Utilization	Remote Miss Delay
Ring	3.59	27.9%	51.5%	1,405ns

Table 8: SIMPLE - 64 nodes

parameter	Normalized Exec. Time	Processor Utilization	Bus/Ring Utilization	Remote Miss Delay
Bus	73.8	1.36%	100.0%	38,410ns

Table 9: WEATHER - 64 nodes

parameter	Normalized Exec. Time	Processor Utilization	Bus/Ring Utilization	Remote Miss Delay
Ring	1.80	55.7%	34.5%	1,201ns
Bus	21.8	4.62%	99.7%	34,576ns

miss rate is 6.0% (29.2% for shared data) for WEATHER, and 16.0% (54.2% for shared data) for SIMPLE.

It is interesting to observe that the bus system is already close to saturation even when the system has only 16 20 MIPS processors. Actually, the 8 processor configuration only shows a fair performance because the average miss rate is very small and hides a very high average shared miss latency. The high miss latency figures for the bus system are a result of contention for the interconnection since the pure latency[3] for a remote miss in this system is less than 400ns. In the ring system, however, the remote miss figures are much closer to the pure remote miss latency, which is consistent with the low ring utilization figures. The pure miss latency on the ring is 260ns. for an 8 node system and 1,100ns for a 64 node system. However, even though a 64 node ring is still far from being saturated, the average processor utilization is never larger than 56%. This behavior suggests that it is the pure latency of accesses that is limiting the overall system performance, and not the interconnection bandwidth. This is an indication that latency tolerance techniques will still be needed, even for very high-bandwidth systems as the Slotted Ring. The only alternative to latency tolerant techniques in such a system is to clock the ring faster and actually reduce the time to traverse the ring. However, even if this were possible, the ring would operate at extremely low utilizations, which is a questionable engineering solution.

It is clear, however, that an architecture such as the Slotted Ring largely outperforms high-end bus systems. In addition, the ring bandwidth can scale up to yet undefined limits, whereas bus systems are not expected to do so. With respect to other point-to-point architectures the Slotted Ring has the benefit of simplicity and efficient use of communication resources. Routing is all but eliminated and forwarding of messages is done with minimal delay.

7 Conclusion

In this paper we have presented a comprehensive approach to enhancing the scalability of shared-memory multiprocessors by reducing the memory penalties. The approach is based on weak ordering and is an alternative to multithreaded processors which have problems of their own. Contrary to the multithreading approach we advocate running one thread per processor and running each thread as fast as possible by overlapping processor execution with multiple outstanding invalidations and misses. In present systems, some overlap is provided by a store buffer and a first level cache; the fact that the cache is blocking is a major handicap. Lockup-free caches provide some latency tolerance for low-to-large latencies by overlapping load misses with processor execution provided the dependency distance can be increased through compiler transformations; for very large latencies the overlap of multiple outstanding load misses by grouping loads together can provide some relief. To overlap multiple outstanding invalidations we have proposed delayed consistency protocols. By delaying consistency we can also reduce drastically false sharing misses, which make write-invalidate cache protocols non-optimum; once the false sharing miss rate is eliminated, the only remaining misses are cold misses, replacement misses and true sharing misses which are inherent to the communication requirements of the parallel application.

Finally, we have evaluated a simple interconnection for multiprocessor: the Slotted Ring, a unidirectional

3. We use the terms *pure latency* to designate the latency in the absence of conflicts

pipelined ring. This interconnection has a large bandwidth and can be packaged easily. It can connect up to 64 powerful processors without running out of bandwidth; however, since the latency grows linearly with the number of processors, latency tolerant techniques presented in this paper should be used to enhance the ring's scalability.

8 Selected References

[1] Dubois, M., Scheurich, C., "Memory Access Dependencies in Shared Memory Multiprocessors," *IEEE Trans. on Soft. Eng.*, 16(6), pp. 660-674, June 1990.

[2] Scheurich, C. *Access Ordering and Coherence in Shared-Memory Multiprocessors.* PhD thesis, Univ. of Southern California, May 1989 (also U.S.C. Tech. Rep. CENG 89-19)

[3] Scheurich, C. and Dubois, M., "Correct Memory Operation of Cache-based Multiprocessors," *Proc. of the 14th Int. Symp. on Comp. Arch*, pp. 234-243, June 1987.

[4] Lenoski, D., et al., "The Directory-based Cache Coherence Protocol for the DASH Multiprocessor," *Proc. of the 17th Ann. Int. Symp. on Comp. Arch., pp. 148-159*, June 1990.

[5] Gharachorloo, K., Gupta, A, and J. Hennessy, "Performance Evaluation of Memory Consistency Models for Shared-memory Multiprocessors," *ASPLOS IV*, Apr 1991.

[6] Kowalik, J. S. *Parallel MIMD Computation: HEP Supercomputer and its Applications.* The MIT Press, 1985.

[7] Dubois, M., "A Cache-based Multiprocessor with High Efficiency," *IEEE Trans. on Comp.*, pp. 968-972, Oct. 1985.

[8] Agarwal, A., et al., "APRIL: A Processor Architecture for Multiprocessing," *Proc. of the 17th Ann. Int. Symp. on Comp. Arch.*, pp. 104-114, June 1990.

[9] Jouppi, N. J., and Wall, D., "Available Instruction-level Parallelism for Superscalar and Superpipelined Machines," ASPLOS III, pp. 272-282, Apr 1989.

[10] Amdahl, G. M., "Validity of the Single Processor Approach to Achieving Large-scale Computing Capabilities," *Proc. AFIPS*, Vol. 30, pp.483-465, 1967.

[11] Scheurich, C. and Dubois, M., "Lockup-free Caches in High-Performance Multiprocessors," *J. of Par. and Dist. Comp.*, Jan. 1991.

[12] Kroft, D., "Lockup-free Instruction Fetch/Prefetch Cache Organization," *Proc. of the 8th Ann. Int. Symp. on Comp. Arch.*, pp. 81-87, June 1981.

[13] Kogge, P. M. *The Architecture of Pipelined Computers.* Mc Graw-Hill, 1981.

[14] McMahon, F. H. LLNL Fortran Kernels: MFlops. Technical Report, Lawrence Livermore Laboratories, Livermore, CA, March 1984.

[15] Zima, H. and Chapman, B. *Supercompilers for Parallel and Vector Computers.* Addison-Wesley Publishing Company. 1990.

[16] Porterfield, A. K. *Software Methods for Improvement of Cache Performance on Supercomputer Applications.* PhD dissertation, RICE COMP TR 89-93, May 1989.

[17] Eggers, S. J., and Jeremiassen, T. E., "Eliminating False Sharing," *Proc. of the 1991 Int. Conf. on Par. Processing*, pp. I-377-I-381, Aug. 1991.

[18] Dubois, M., et al., "Delayed Consistency and its Effects on the Miss Rate of Parallel Programs," *Supercomputing'91*, pp. 197-206, Nov. 1991.

[19] Censier, L. M. and Feautrier, P., "A New Solution to Coherence Problems in Multicache Systems," *IEEE Trans. on Computers*, Vol. C-27, No. 12, pp. 1112-1118, Dec. 1978.

[20] Gustavson, D. B., "The Scalable Coherent Interface and Related Standards Projects", *IEEE Micro*, Vol. 12, No. 1, pp. 10-22, February 1992.

[21] Barroso, L. A. and Dubois, M., "Cache Coherence on a Slotted Ring", *Proceedings of the 1991 International Conference on Parallel Processing*, pp. I230-I237, August 1991.

[22] Schwetman, H., "CSIM: A C-Based, Process-Oriented Simulation Language", *Proceedings of the 1986 Winter*